Available online at www.sciencedirect.com
SCIENCE DIRECT®
Theoretical Computer Science 335 (2005) 187–213
Theoretical Computer Science
www.elsevier.com/locate/tcs

Measuring the performance of asynchronous systems with PAFAS

F. Corradini[a,1,*], W. Vogler[b]

[a] *Dipartimento di Matematica e Informatica, Università di Camerino,Via del Bastione 3, Camerino 62032, Italy*
[b] *Institut für Informatik, Universität Augsburg, Germany*

Received 13 November 2003; accepted 30 January 2004

Abstract

Based on Process Algebra for Faster Asynchronous Systems (PAFAS), a testing-based faster-than relation has previously been developed that compares the worst-case efficiency of asynchronous systems. This approach reveals that pipelining does not improve efficiency in general; that it does so in practice depends on assumptions about the user behaviour. As a case study for testing under such assumptions, we adapt the PAFAS-approach to a setting where user behaviour is known to belong to a specific, but often occurring class of request–response behaviours.

Just as the testing preorder in classical testing, the original faster-than relation is qualitative. We give it a quantitative reformulation for the general approach; based on this, we demonstrate in our case study how to determine an asymptotic performance measure for finite-state processes. With this result, we can show that pipelining indeed improves efficiency in our setting, and we discuss additional examples.

Keywords: Concurrent systems; Process algebra; Timed testing; Efficiency

1. Introduction

Recently, Process Algebra for Faster Asynchronous Systems (PAFAS) has been proposed as a useful tool for comparing the worst-case efficiency of asynchronous systems [2]. PAFAS

* Corresponding author.
E-mail addresses: flavio.corradini@unicam.it (F. Corradini), vogler@informatik.uni-augsburg.de (W. Vogler).

[1] This work of F. Corradini was supported by MURST project 'Sahara: Software Architectures for Heterogeneous Access Networks infrastructures' and by the Center of Excellence for Research 'DEWS: Architectures and Design Methodologies for Embedded Controllers, Wireless Interconnect and System-on-chip'.

0304-3975/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2004.01.039

is a CCS-like process description language [10] where a basic action is atomic and instantaneous but has an associated time bound specifying the maximal delay for its execution. (For simplicity, these bounds are always 1 or 0.) As discussed in [2], these upper time bounds give information on the efficiency of processes, but in contrast to most timed process algebras, time does not influence the functionality (i.e. which actions are performed); so like CCS, also PAFAS treats the full functionality of asynchronous systems. Processes are compared via a variant of the testing approach [3] where a (timed) test consists of a test environment (or user behaviour) together with a time bound. A process is embedded into the environment (via parallel composition) and satisfies the test, if success is reached before the time bound in *every* run of the composed system, i.e. even in the worst case. This gives rise to a preorder relation over processes which is naturally an *efficiency* or *faster-than preorder*. This efficiency preorder can be characterized as inclusion of some kind of refusal traces. It has also been shown that the preorder is independent of the choice to regard time as continuous or discrete; therefore, we only consider discrete time in this paper. These ideas and results were originally successfully studied in the Petri net formalism [12,5]. We refer the reader to [2] for more details and results on PAFAS.

Consider a task (like a processing of some data) that can be performed in two stages (e.g. the compilation of a programme). In a sequential architecture, the process performs both stages for such a task and only then starts with the next task. In PAFAS, we can model this process as $\mathsf{Seq} \equiv \mu x.\underline{in}.\tau.out.x$, where *in* is the input of data or some other request (the underbar indicating time bound 0), τ is an internal activity corresponding to the first stage, and the second stage is integrated into *out*, which outputs the result or gives a response to the request and takes time up to 1 as the first stage. Alternatively, one could use a pipelined architecture, where a second task can be accepted already when the first stage is over for the first task. This process is $\mathsf{Pipe} \equiv ((\mu x.\underline{in}.s.x) \|_{\{s\}} (\mu x.\underline{s}.out.x))/s$, where the first processing stage is integrated into the shift-action *s* in the first component.

Even though these are asynchronous systems, where the times needed by actions are not exactly known, one would expect that Pipe is faster than Seq since it allows more parallelism. But it turns out that Pipe is not faster for the following reason: in the PAFAS-approach, if one system is faster than another, it also functionally refines this other system as in ordinary testing; in particular, it cannot perform new action sequences—but Pipe can perform the sequence $in\,in$, which is not possible for Seq.

Obviously, it is important to reconcile a theory for efficiency with the expectation that the general principle of pipelining improves efficiency. The argument of the previous paragraph reveals that the expectation of Pipe being faster is based on some assumptions about the users, e.g. that their coordination will not be disturbed by the new action sequence $in\,in$. While a testing approach that considers all possible test environments or contexts usually leads to a precongruence, this cannot be expected in a test setting with a restricted class of users (or test environments), and it is not immediately clear what sort of results one can achieve. This paper can be seen as a case study for such a practical situation, and with the results of this case study we will show that pipelining indeed improves efficiency in a restricted setting.

The idea to study testing with an essentially restricted class of users has certainly been considered before; but we do not know of any published results. Thus, we regard it as quite an achievement that we can give useful results for a practical approach of this type.

To formulate our results, we start with a general observation: the testing approach is qualitative since a timed test is either satisfied or not and one system is faster than another or not. But often a quantitative performance measure seems more attractive: it would say how much faster the first system is, i.e. how valuable it is compared to the second system; also, if we do not have a second system as reference model, we would still like to have an idea of our system's performance. Therefore, we point out that the faster-than preorder of Corradini, Vogler and Jenner [2] can equivalently be defined on the basis of a *performance function* that gives for each user behaviour the worst-case time needed to satisfy the user.

Then, we will adapt the timed testing scenario as announced above by considering only users (or test environments) U_n that want n tasks to be performed as fast as possible, i.e. possibly in parallel; these users correspond to the practically important scenario where systems are tested under heavy load. Given a process, the performance function assigns to each U_n the time it takes in the worst case to satisfy U_n, i.e. it is essentially a function from natural numbers to natural numbers. Since it measures how fast the system under consideration responds to requests, we call this function the *response performance* of the system. For finite-state processes that are functionally correct in a sense to be defined (cf. Definition 10), we prove that the response performance is asymptotically linear, and we show how to determine its factor, which we call the *asymptotic performance*. Such a result is not so unusual in performance evaluation in general, but as far as we know we obtain it in an unusual setting.

As an application of this new response testing scenario, one finds that the asymptotic performance is 1 for Pipe and 2 for Seq—justifying the expectation that pipelining increases efficiency. For both processes, the worst case is assumed when each action takes as long as possible, i.e. it is assumed for runs where times are exact and not just upper bounds, as if the systems run in synchronous mode. In Section 4 we show another example, which demonstrates that it can be a serious mistake to consider only such synchronous runs: for this example process, the asymptotic performance would be 1 if runs were synchronous. However, taking all asynchronous behaviour into account shows that the asymptotic performance is in fact at least 1.5. Observe that making a system synchronous requires the additional effort of introducing a global clock signal; we give a small variation of the process that really has asymptotic performance 1.

Section 2 briefly recalls PAFAS, the timed testing scenario and the alternative characterization in terms of refusal traces; see [2] for more detailed explanations. It also presents the new quantitative formulation of the testing definition. We adapt the timed testing scenario to the new response testing in Section 3, and we show the new results about the performance function and the asymptotic performance. The examples are studied in Section 4 and, in Section 5, we give a conclusion and discuss related work.

2. PAFAS

In this section, we briefly introduce our CCS-like process description language PAFAS (with a TCSP-like parallel composition), its operational semantics and a testing-based preorder relating processes according to the worst-case efficiency. For an easier presentation,

we will define the operational semantics of PAFAS using refusal sets; then the testing preorder is based on this semantics.

In this presentation, it does not look so surprising that this preorder can be characterized with refusal traces. In [2], the operational semantics is defined in a way which is closer to intuition and independent of refusal sets, but more complicated; there, the characterization result looks more surprising, but actually its proof would not be so much easier with the definitions used here. We refer the reader to [2] for more details.

We close this section with a quantitative reformulation of our testing scenario.

- $\mathbb{A}$ is an infinite set of actions $a, b, c, \ldots$ with the special action ω, which is reserved for observers (test processes) in the testing scenario to signal success of a test. The additional action τ represents internal activity that is unobservable for other components. We define $\mathbb{A}_\tau = \mathbb{A} \cup \{\tau\}$ (ranged over by $\alpha, \beta, \ldots$). Actions in $\mathbb{A}_\tau$ can let time 1 pass before their execution, i.e. 1 is their maximum delay. After that time, they become *urgent* actions. The set of urgent actions is denoted by $\underline{\mathbb{A}}_\tau = \{\underline{a} \mid a \in \mathbb{A}\} \cup \{\underline{\tau}\}$ and is ranged over by $\underline{\alpha}, \underline{\beta}, \ldots$ (In most cases, longer delays can be specified by additional τ-prefixes.)
- χ is the set of process variables $x, y, z, \ldots$, used for recursive definitions.
- Take a function $\Phi : \mathbb{A}_\tau \to \mathbb{A}_\tau$ such that the set $\{\alpha \in \mathbb{A}_\tau \mid \emptyset \neq \Phi^{-1}(\alpha) \neq \{\alpha\}\}$ is finite, $\Phi^{-1}(\omega) \subseteq \{\omega\}$ and $\Phi(\tau) = \tau$; then Φ is a *general relabelling function*. As shown in [2], general relabelling functions subsume the classically distinguished operations relabelling and hiding: P/A, where the actions in A are made internal, is the same as $P[\Phi_A]$, where the relabelling function Φ_A is defined by $\Phi_A(\alpha) = \tau$ if $\alpha \in A$ and $\Phi_A(\alpha) = \alpha$ if $\alpha \notin A$.

Definition 1 (*Timed and initial processes*). The set $\tilde{\mathbb{P}}$ of *(discretely timed) process terms* is the set of terms generated by the following grammar:

$$P ::= 0 \mid \alpha.P \mid \underline{\alpha}.P \mid P + P \mid P \|_A P \mid P[\Phi] \mid x \mid \mu x.P,$$

where $x \in \chi$, $\alpha \in \mathbb{A}_\tau$, Φ a general relabelling function, $A \subseteq \mathbb{A}$ possibly infinite and recursion is time-guarded, i.e. variable x in $\mu x.P$ only appears within the scope of an $\alpha.()$-prefix with $\alpha \in \mathbb{A}_\tau$. $\mathbb{P}$ is the set of closed terms (i.e. without free variables), called *processes*.

$\mathbb{P}_1$ is the set of so-called *initial processes*, i.e. processes where all actions are from set $\mathbb{A}_\tau$. This is a significant subset of $\mathbb{P}$ since it corresponds to ordinary CCS-like processes.

0 is the Nil-process, which cannot perform any action, but may let time pass without limit; a trailing 0 will often be omitted, so e.g. $a.b + c$ abbreviates $a.b.0 + c.0$. $\alpha.P$ and $\underline{\alpha}.P$ is (action-)prefixing, known from CCS. Process $\alpha.P$ performs α with a *maximal* delay of 1; hence, it can perform α immediately or it can idle for one time unit and become $\underline{\alpha}.P$. As a stand-alone process, $\underline{\alpha}.P$ must perform α immediately; as a component, it may also be deactivated in a choice-context or it may have to wait for synchronization with another component in a parallel context (in case $\alpha \neq \tau$). This means that our processes are *patient*: as a stand-alone process, $\underline{a}.P$ has no reason to wait; but as a component in $(\underline{a}.P) \|_{\{a\}} (a.Q)$, it has to wait for synchronization on a and this can take up to time 1, since component $a.Q$

may idle this long. That a process may perform a conditional time step, i.e. may take part in a time step in certain contexts, is the intuition behind refusal sets defined below.

$P_1 + P_2$ models the choice (sum) of two conflicting processes P_1 and P_2. $P_1 \|_A P_2$ is the parallel composition of two processes P_1 and P_2 that run in parallel and have to synchronize on all actions from A; this synchronization discipline is inspired from *TCSP*. $P[\Phi]$ behaves as P but with the actions changed according to Φ. $\mu x.P$ models a recursive definition.

Now, the purely functional behaviour of processes (i.e. which actions they can perform) is given by the following operational semantics.

Definition 2 (*Operational semantics of functional behaviour*). The following SOS-rules define the transition relations $\xrightarrow{\alpha} \subseteq \tilde{\mathbb{P}} \times \tilde{\mathbb{P}}$ for $\alpha \in \mathbb{A}_\tau$, the *action transitions*. As usual, we write $P \xrightarrow{\alpha} P'$ if $(P, P') \in \xrightarrow{\alpha}$ and $P \xrightarrow{\alpha}$ if there exists a $P' \in \tilde{\mathbb{P}}$ such that $P \xrightarrow{\alpha} P'$, and similar conventions will apply later on.

$$\text{Pref}_{a1}\ \frac{}{\alpha.P \xrightarrow{\alpha} P}, \qquad \text{Pref}_{a2}\ \frac{}{\underline{\alpha}.P \xrightarrow{\alpha} P},$$

$$\text{Par}_{a1}\ \frac{\alpha \notin A,\ P_1 \xrightarrow{\alpha} P_1'}{P_1 \|_A P_2 \xrightarrow{\alpha} P_1' \|_A P_2}, \qquad \text{Par}_{a2}\ \frac{\alpha \in A,\ P_1 \xrightarrow{\alpha} P_1',\ P_2 \xrightarrow{\alpha} P_2'}{P_1 \|_A P_2 \xrightarrow{\alpha} P_1' \|_A P_2'},$$

$$\text{Sum}_a\ \frac{P_1 \xrightarrow{\alpha} P_1'}{P_1 + P_2 \xrightarrow{\alpha} P_1'}, \qquad \text{Rel}_a\ \frac{P \xrightarrow{\alpha} P'}{P[\Phi] \xrightarrow{\Phi(\alpha)} P'[\Phi]},$$

$$\text{Rec}_a\ \frac{P \xrightarrow{\alpha} P'}{\mu x.P \xrightarrow{\alpha} P'\{\mu x.P/x\}}.$$

Additionally, there are symmetric rules for Par_{a1} and Sum_a for actions of P_2.

Finally, $\mathcal{A}(P) = \{\alpha \in \mathbb{A}_\tau \mid P \xrightarrow{\alpha}\}$ is the set of *activated actions* of P.

Except for Pref_{a2}, these rules are standard. Pref_{a1} and Pref_{a2} allow an activated action to occur (just as e.g. in CCS), and it makes *no* difference whether the action is urgent or not. Additionally, passage of time will never deactivate actions or activate new ones, and we capture all behaviour that is possible in the standard CCS-like setting without time.

That our SOS-rules describe asynchronous behaviour is due to Pref_{a1}; this rule allows to ignore the possible delay of α, and thus timing cannot be used to coordinate system behaviour. We get *synchronous* behaviour, if we do not use Pref_{a1}, because then process $\alpha.P$ has to delay exactly one time unit, after which α becomes enabled and urgent at the same time. We will exemplify this after the definition of time steps.

The set of activated actions $\mathcal{A}(P)$ of a process P describes its immediate functional behaviour. It records only actions, ignoring whether they are urgent or not, and is finite as proven in [2].

We now give SOS-rules for so-called *refusal sets*. Performing such a set X is a conditional time step (of duration 1) and X consists of (some, but not necessarily all) actions which are *not* just waiting for synchronization; i.e. these actions are *not* urgent, the process does not have to perform them at this moment, and they can therefore be refused. If a process can perform

a conditional time step, then it can take part in a 'real' time step in a suitable environment; the refusal set describes requirements for such an environment and the conditional time step also describes the effect on the process.

For e.g., according to rule Pref_{r2} below, $\underline{\alpha}.P$ with $\alpha \in \mathbb{A}$ can refuse all actions except α. As explained after Definition 1, this process cannot perform a time step as a stand-alone process, but in a parallel composition it might take part in a time step, if the other component can refuse α; in this example, the process does not change in this step. Note that $\underline{\tau}.P$ has to perform its urgent τ before the next time step—independently of the environment—, hence this process cannot perform a refusal set. If a process can refuse all actions, it can indeed perform a time step also as a stand-alone process.

Definition 3 (*SOS-rules for refusal sets*). The following SOS-rules define $\xrightarrow{X}_r \subseteq \tilde{\mathbb{P}} \times \tilde{\mathbb{P}}$, where $X, X_i \subseteq \mathbb{A}$:

$$\mathrm{Nil}_r \ \frac{}{0 \xrightarrow{X}_r 0}, \qquad \mathrm{Pref}_{r1} \ \frac{}{\alpha.P \xrightarrow{X}_r \underline{\alpha}.P}, \qquad \mathrm{Pref}_{r2} \ \frac{\alpha \notin X \cup \{\tau\}}{\underline{\alpha}.P \xrightarrow{X}_r \underline{\alpha}.P},$$

$$\mathrm{Par}_r \ \frac{\forall_{i=1,2}\ P_i \xrightarrow{X_i}_r P_i',\ X \subseteq \left(A \cap \bigcup_{i=1,2} X_i\right) \cup \left(\left(\bigcap_{i=1,2} X_i\right) \setminus A\right)}{P_1 \|_A P_2 \xrightarrow{X}_r P_1' \|_A P_2'},$$

$$\mathrm{Sum}_r \ \frac{\forall_{i=1,2}\ P_i \xrightarrow{X}_r P_i'}{P_1 + P_2 \xrightarrow{X}_r P_1' + P_2'}, \qquad \mathrm{Rel}_r \ \frac{P \xrightarrow{\Phi^{-1}(X \cup \{\tau\}) \setminus \{\tau\}}_r P'}{P[\Phi] \xrightarrow{X}_r P'[\Phi]},$$

$$\mathrm{Rec}_r \ \frac{P \xrightarrow{X}_r P'}{\mu x.P \xrightarrow{X}_r P'\{\mu x.P/x\}}.$$

When $P \xrightarrow{X}_r P'$, we call this a (conditional) *time step* or, if $X = \mathbb{A}$, a *full time step*. In the latter case, we also write $P \xrightarrow{1} P'$.

For example, $\underline{a}.P$ ($a.Q$, resp.) can make a time step with refusal set $\mathbb{A} \setminus \{a\}$ (with refusal set $\mathbb{A}$, resp.) according to rule Pref_{r2} (Pref_{r1}, resp.) and with rule Par_r we get $(\underline{a}.P)\|_{\{a\}}(a.Q) \xrightarrow{1} (\underline{a}.P)\|_{\{a\}}(\underline{a}.Q)$ as announced above. Similarly, the process $\mathsf{Pipe} \equiv ((\mu x.\underline{in}.s.x)\|_{\{s\}}(\mu x.\underline{s}.out.x))/s$ from the introduction can make a conditional time step with refusal set $\mathbb{A} \setminus \{in\}$, meaning it can take part in a full time step provided the environment does not offer an urgent *in*. This time step leads essentially back to Pipe, but it results in an unfolding of the two recursions. Note that the second parallel component of Pipe has an urgent *s*, but this is refused by the first (which cannot perform *s* at all).

Pipe can also perform *in* leading to $\mathsf{Pipe}' \equiv ((s.\mu x.\underline{in}.s.x)\|_{\{s\}}(\mu x.\underline{s}.out.x))/s$. Now the communication partner for the urgent *s* of the second parallel component is activated, but not urgent. Formally, the first component can refuse $\mathbb{A}$ according to Pref_{r1}, while the second can refuse $\mathbb{A} \setminus \{s\}$ according to Pref_{r2}; their composition can refuse $\mathbb{A}$ according to Par_r, since it is enough that one component can refuse *s*, which gets synchronized. Therefore (according to Rel_r), Pipe′ can perform a full time step corresponding to the first processing stage and leading to the process $((\underline{s}.\mu x.\underline{in}.s.x)\|_{\{s\}}(\underline{s}.out.\mu x.\underline{s}.out.x))/s$. This process in turn can perform an urgent τ resulting from *s* and, thus, no time step is possible. The τ leads

to a process $((\mu x.\underline{in}.s.x)\|_{\{s\}}(out.\mu x.\underline{s}.out.x))/s$, which can also be reached directly from Pipe' with a τ. Further observe the effect of inserting a τ-prefix in front of s in Pipe; this gives an upper time bound of 2 for the first stage.

We now give an example to demonstrate the difference between synchronous behaviour and the behaviour we consider here. Consider $P \equiv (a.b)\|_{\{b\}}(b + \tau.c)$; we will explore whether P can perform action b. According to the above definitions we have $P \xrightarrow{a} \xrightarrow{b} 0\|_{\{b\}}0$. With synchronous behaviour, i.e. without using rule Pref_{a1}, no action is possible for P; hence, only a time step $P \xrightarrow{X}_r (\underline{a}.b)\|_{\{b\}}(\underline{b} + \underline{\tau}.c)$ can be performed. To perform b, the latter process must make an a-transition to $b\|_{\{b\}}(\underline{b} + \underline{\tau}.c)$; here, the first component cannot perform b in synchronous mode, hence no b can be performed; and also a time step is not possible due to the urgent τ. Thus, the only transition is a τ-transition to $b\|_{\{b\}}c$, which clearly will never perform b.

Both, purely functional and timed behaviour of processes will now be combined in the language and in the refusal traces of processes. The language of P is its behaviour as a stand-alone process; such a process never has to wait for a communication, hence all time steps in a run are full. As usual, we will abstract from internal behaviour; but note that internal actions gain some 'visibility' in timed behaviour, since their presence possibly allows more time to pass in between the occurrence of visible actions.

Definition 4 (*Language, refusal traces*). Let $P, P' \in \mathbb{P}$ be processes. We extend the transition relation $P \xrightarrow{\mu} P'$ for $\mu \in \mathbb{A}_\tau$ or $\mu = 1$ to sequences w and write $P \xrightarrow{w} P'$ if $P \equiv P'$ and $w = \varepsilon$ (the empty sequence) or there exist $Q \in \mathbb{P}$ and $\mu \in \mathbb{A}_\tau \cup \{1\}$ such that $P \xrightarrow{\mu} Q \xrightarrow{w'} P'$ and $w = \mu w'$.

For a sequence $w \in (\mathbb{A}_\tau \cup \{1\})^*$, let w/τ be the sequence w with all τ's removed, and let the *duration* $\zeta(w)$ of w be the number of full time steps in w; note that $\zeta(w/\tau) = \zeta(w)$. We write $P \overset{v}{\Rightarrow} P'$, if $P \xrightarrow{w} P'$ and $v = w/\tau$. Now we define $\mathsf{DL}(P) = \{w \mid P \overset{w}{\Rightarrow}\}$ to be the *(discretely timed) language*, containing the *(discrete) traces* of P.

The *timed transition system* $\mathsf{TTS}(P)$ of P consists of all transitions $Q \xrightarrow{\mu} Q'$ with $\mu \in \mathbb{A}_\tau$ or $\mu = 1$ where Q is reachable from P via such transitions.

For processes $P, P' \in \mathbb{P}$, we similarly write $P \xrightarrow{\mu}_r P'$, if either $\mu = \alpha \in \mathbb{A}_\tau$ and $P \xrightarrow{\alpha} P'$, or $\mu = X \subseteq \mathbb{A}$ and $P \xrightarrow{X}_r P'$. For sequences w, we define $P \xrightarrow{w}_r P'$ and $P \overset{w}{\Rightarrow}_r P'$ as above. $\mathsf{RT}(P) = \{w \mid P \overset{w}{\Rightarrow}_r\}$ is the set of *refusal traces* of P. We write $P \leqslant_r Q$ if $\mathsf{RT}(P) \subseteq \mathsf{RT}(Q)$.

The *refusal transition system* $\mathsf{RTS}(P)$ of P consists of all transitions $Q \xrightarrow{\mu}_r Q'$ with $\mu \in \mathbb{A}_\tau$ or $\mu \subseteq \mathbb{A}$ where Q is reachable from P via such transitions. If $\mathsf{RTS}(P)$ contains only finitely many processes, we call P *finite state*.

Note that $\mathsf{RTS}(P \parallel_A Q)$ can be determined from $\mathsf{RTS}(P)$ and $\mathsf{RTS}(Q)$ according to the SOS-rules for parallel composition given above. $\mathsf{TTS}(P)$ can be obtained from $\mathsf{RTS}(P)$ by deleting time steps that are not full and processes that then are not reachable anymore.

By Proposition 5.1 below, the set of possible refusal sets for a process is downward closed w.r.t. set inclusion, and by Item 2, non-activated actions can always be refused. Hence, only

the refusal of activated actions is relevant to determine the time steps of a process. Item 3 shows that time steps do not enable new behaviour, corresponding to our idea of asynchronous behaviour. Item 4 states that PAFAS-processes do not have time-stops, i.e. any process can perform any number of full time steps; hence, time can always proceed ad infinitum, which is to be expected intuitively. For the proofs we refer to [2]; in particular, see Proposition 5.12 for Item 3.

Proposition 5. *Let $P, Q, R \in \mathbb{P}$ be processes and let $X, X' \subseteq \mathbb{A}$.*
(1) *If $P \xrightarrow{X}_r Q$ and $X' \subseteq X$, then $P \xrightarrow{X'}_r Q$.*
(2) *If $P \xrightarrow{X}_r Q$ and $X' \cap \mathcal{A}(P) = \emptyset$, then $P \xrightarrow{X \cup X'}_r Q$.*
(3) *If $wXw' \in \mathsf{RT}(P)$, then $ww' \in \mathsf{RT}(P)$.*
(4) *For all $n \in \mathbb{N}$, there is some w with $P \xrightarrow{w}$ and $\zeta(w) > n$.*

Based on the language of processes, we are now ready to define timed testing and to relate processes w.r.t. their efficiency, thereby defining an *efficiency preorder*:

Definition 6 (*Timed tests*). A process $P \in \mathbb{P}$ is *testable* if ω does not occur in P. Any process $O \in \mathbb{P}$ may serve as a *test process* (*observer*). We write $\|$ for $\|_{\mathbb{A}\setminus\{\omega\}}$.

A *timed test* is a pair (O, D), where O is a test process and $D \in \mathbb{N}_0$ is the *time bound*. A testable process P *d-satisfies* such a timed test (P *must*$_d$ (O, D)), if each $v \in \mathsf{DL}(P\|O)$ with $\zeta(v) > D$ contains some ω.

For testable processes P and Q, we call P a *faster implementation* of Q or *faster than* Q, written $P \sqsupseteq Q$, if P d-satisfies all timed tests that Q d-satisfies.

Runs with duration less than D may not contain all actions that occur up to time D; hence we only consider runs with a duration greater than the time bound D for test satisfaction. The operational idea behind this is that—when performing a test—one should certainly wait until time D is up before declaring the test a failure. By definition, $P \sqsupseteq Q$ means that P is functionally a refinement of Q, since it is satisfactory for at least as many test processes as Q, and that it is an improvement timewise, since it d-satisfies test processes at least as fast as Q.

Corradini, Vogler and Jenner [2] actually only consider initial processes O as test process and, related to this, embed P in the form $\tau.P\|O$ instead of $P\|O$. Thus, the proof of the following result has to be adapted accordingly—by turning the 'initial' actions of the test processes given in [2] into urgent actions. For this result, recall that it looks more surprising in the setting of Corradini et al. [2], where the language of a process is defined independently of refusal traces; still, also in our present setting, the following result is in no way straightforward to prove. Indeed, the result states that timed tests can see refusal traces, which give quite a detailed account of the timed behaviour of processes; this is quite surprising, since we are in an asynchronous setting, where tests should have little temporal control over the tested systems.

Obviously, it is impossible to apply the definition of faster-than directly, since there are already countably many time bounds and, hence, timed tests to apply. Hence, it is very helpful that the efficiency preorder can be characterized by refusal-trace-inclusion.

Theorem 7 (*Characterization of the testing preorder*). *Let P, Q be testable processes. Then $P \sqsupseteq Q$ if and only if $P \leqslant_r Q$.*

If P is not faster than Q, i.e. $P \not\sqsupseteq Q$, then there is a refusal trace of P that is not one of Q. Roughly speaking, this is a witness of slow behaviour of P; it is a diagnostic information that tells us why P is not faster. If P and Q are finite-state, inclusion of refusal traces can be checked automatically; a corresponding tool, FastAsy, has been developed for a Petri net setting [1], and adaptation to PAFAS is in progress. In case that P is not faster, FastAsy presents a refusal trace as witness; this can be used to improve P—and in practice, it can also help to find errors that can occur when formalizing an intuitive idea as a PAFAS-process.

Witnesses of slow behaviour will also play an important role in the next section in the form of what we will call n-critical paths.

As usual, our testing scenario is qualitative in the sense that a timed test is either satisfied or not. We now give an easy, but new reformulation of the scenario that brings to light its quantitative nature.

Definition 8 (*Performance*). For a testable process $P \in \mathbb{P}$ and test process $O \in \mathbb{P}$, we define the *performance function* p by

$$p(P, O) = \sup\{n \in \mathbb{N}_0 \mid \exists v \in \mathsf{DL}(P \| O) : \zeta(v) = n \text{ and } v \text{ does not contain } \omega\}.$$

If the set on the right-hand side has no maximum, the supremum is ∞. The *performance function p_P of P* is defined by $p_P(O) = p(P, O)$.

Proposition 9. *Let P, $Q \in \mathbb{P}$ be testable processes. Then $P \sqsupseteq Q$ if and only if for all test processes O we have $p(P, O) \leqslant p(Q, O)$, i.e. $p_P \leqslant p_Q$.*

Proof. P is not faster than Q iff, for some timed test (O, D), Q d-satisfies (O, D) and P does not iff $p(Q, O) \leqslant D < p(P, O)$. □

3. Asymptotic performance

3.1. Response performance

The example discussed in the introduction has demonstrated that in some cases there are assumptions on the user behaviour, i.e. one is only interested in test processes or users from a certain class $\mathcal{U}$. In this case, one should compare the performance functions of some P and Q only for arguments from $\mathcal{U}$ to determine which of the two is faster.

In some cases, one may be able to group the users in $\mathcal{U}$ according to their 'size' into disjoint classes $\mathcal{U}_i$, $i = 1, 2, \ldots$; then, one can turn the performance function of P into a function from $\mathbb{N}$ to $\mathbb{N}_0 \cup \{\infty\}$ (which we will call rp_P below) that assigns to each i the value $\sup\{p_P(O) \mid O \in \mathcal{U}_i\}$. This is the sort of worst-case efficiency measure we are used to from 'ordinary' algorithms. Since $p_P(O)$ can be determined from $\mathsf{TTS}(P \parallel O)$, which in turn can be determined from $\mathsf{RTS}(P)$ and $\mathsf{RTS}(O)$, it is in principle sufficient to consider $\mathsf{RTS}(P)$ to find out interesting facts about rp_P; for this to be feasible, the O under consideration

must presumably be 'uniform' enough. To obtain effective results, we will restrict ourselves later to finite state P.

We will demonstrate this approach with a specific class of users that will in particular allow to compare the processes Pipe and Seq from the introduction in such a way that the expected result holds and pipelining is justified as a strategy to improve efficiency. Users of these processes issue requests with action *in* and expect responses via action *out*. In practice, these actions will usually transfer data, but we assume that we can abstract from these data; this has two aspects: we assume that the correctness of the data produced and output by the processes is checked by some other means, while we are only concerned with efficiency; and we assume that processing times are data-independent—which is not unusual in algorithm analysis. We discuss this in detail in the remark below.

We assume further that the only users of interest have a number of requests that they want to be answered as fast as possible, i.e. possibly in parallel, without any restriction; thus, we consider the users U_n defined by

$$U_1 \equiv \underline{in}.\underline{out}.\underline{\omega},$$

$$U_{n+1} \equiv U_n \parallel_\omega \underline{in}.\underline{out}.\underline{\omega}.$$

Comparing processes w.r.t. these users means to compare their performance under heavy load; this is clearly of practical importance. Cf. also [9, p. 645], where Nancy Lynch discusses the problem of determining the performance for a specific MUTEX-solution—modelled as a reactive asynchronous system with upper time bounds; she resorts to give results for various user behaviours, one of them being the heavy load case.

Remark. Let us discuss the abstraction we make in detail. In general, one would take a more concrete view and consider systems where each request *in* is accompanied by some data x, i.e. it is an action in_x, and the corresponding response additionally gives some result, i.e. it is an action $out_{x,f(x)}$. In this case, a (more concrete) user with a single request (corresponding to our more abstract U_1) would be modelled as

$$\sum_x \underline{\tau}.\underline{in_x}.\left(\sum_y \underline{out_{x,y}}.P_{x,y}\right),$$

where $P_{x,y}$ is $\underline{\omega}$ if $y = f(x)$ and 0 otherwise, and x and y range over some finite data domains such that $\sum_x$ and $\sum_y$ abbreviate choices between finitely many alternatives. The initial $\underline{\tau}$'s model that the user decides which data to submit, independently of the system. Users with several requests could be defined analogously as above.

Under the following two assumptions on the system, one can abstract from data (i.e. replace in_x by *in* and $out_{x,f(x)}$ by *out*) in the system and use our approach to determine worst-case performance. First, inputs must be data-independent in the sense that, whenever some in_x is enabled, then all in_x are enabled—i.e. if the system is able to accept the input of some data, it is also able to accept any other data the user might choose with one of the initial $\underline{\tau}$'s shown above. Second, the system must be functionally correct in the sense that each request in_x is answered by a suitable response $out_{x,f(x)}$, and no responses are generated without request. Functional correctness can be checked disregarding time, since we deal with asynchronous systems where time does not influence functionality; this check

does not concern us here. Thus, our approach is more general than it might seem at first sight.

Clearly, the size of U_n is its number n of requests. Hence, for the classes discussed above, we take $\mathcal{U}_i = \{U_i\}$ and accordingly define the *response performance* rp_P of a testable process P as the function from $\mathbb{N}$ to $\mathbb{N}_0$ with $rp_P(n) = p_P(U_n)$. Our aim is to evaluate (to some degree) the response performance of a process from its refusal transition system. This system is an arc-labelled graph, an arc (or directed edge) being a transition; as usual, a *path* is a sequence of transitions, each ending in a process from which the next transition starts, it is *closed* if the last and first process coincide. If apart from the latter coincidence all processes on a closed path are different, it is a *cycle*. Note that a finite transition system can only have finitely many different cycles. (We explain below, why in our setting a finite-state process essentially has a finite refusal or timed transition system.)

3.2. Response processes

Our results on response performance only hold for processes that can reasonably serve the users U_n, and such processes will be called response processes. As we discuss after their definition, these processes may fail to satisfy some liveness property, which can be seen as concerning time; we come back to correct response processes, which will satisfy this property, when we give our results on performance. In this subsection, we show how to decide whether a testable process is a response process, in the next how to decide its correctness.

Since for the performance of a testable process P, $P \parallel U_n$ is relevant where synchronization is over all visible actions except ω, we assume that each process whose performance we try to evaluate can only perform *in* and *out* as visible actions; other actions would be disallowed by the composition anyway. We restrict the processes under consideration even further as follows:

Definition 10. The *o*-number of a process Q is the number of pending *out* actions, i.e. it is $\sup\{\text{number of } out \text{ in } w \mid Q \stackrel{w}{\rightarrow}_r \text{ and } w \text{ does not contain } in\}$.

A testable process is a *response process* if it can only perform *in* and *out* as visible actions and is functionally correct in the following sense: if $P \stackrel{w}{\rightarrow}_r Q$, then the number of *in* in w minus the number of *out* in w is non-negative and the *o*-number of Q.

Thus, a response process P is always able to perform the required number of *out* actions and never performs too many. Note that there is a gap in this understanding of functional correctness: although a complying process is *able* to perform the required number of *out*'s, it is not ensured that it *will do so* in a bounded time. Also, it is not sure that a response process will allow another *in* in a bounded time. The property that a process will eventually perform a missing *out* and eventually allow another *in* is the liveness property mentioned above.

In both these cases of incorrect behaviour, the response performance would be ∞ for some n. The latter means that some user will not be satisfied within any time bound, which is certainly an incorrect behaviour of the process. Since this point is concerned with time, we only define correct response processes here and deal with them later.

Definition 11. A response process is *correct* if its response performance is finite for all n.

When constructing $\mathsf{RTS}(P)$ for some P which is supposed to be a response process, we can check on the fly whether P can perform a visible action different from *in* and *out*; if so, we can stop the construction. Otherwise, whenever a time step is performed in $\mathsf{RTS}(P)$ we can add to or remove from the refusal set arbitrary actions in $\mathbb{A} \setminus \{in, out\}$ by Proposition 5.2 and 5.1. Therefore, there are only four significant refusal sets, which for notational convenience we write as $\mathbb{A}$, $\{out\}$, $\{in\}$ and $\emptyset$. When we speak of $\mathsf{RTS}(P)$ in the following, we are referring to this slightly reduced version, which we will reduce even further below. Consequently, if P is *finite state*, $\mathsf{RTS}(P)$ also has finitely many transitions and is a *finite transition system*.

Theorem 12. *Let $P \in \mathbb{P}$ be a testable process, Q reachable from P with o-number o and $Q \xrightarrow{\mu}_r Q'$.*

(1) *Let P be a response process. Then o is finite. Furthermore, if μ is in, out, resp., then the o-number of Q' is $o+1$, $o-1$, resp.; for all other cases of μ, it is o. The numbers of in's and of out's on a closed path in $\mathsf{RTS}(P)$ are equal.*

(2) *If P is finite state, then it is decidable in time linear in the size of $\mathsf{RTS}(P)$ whether P is a response process.*

Proof. (1) If $P \xrightarrow{w}_r Q$, then o must be the number of *in* in w minus the number of *out* in w by the definition of a response process, hence it is finite. This argument also implies the next statement, since $P \xrightarrow{w\mu}_r Q'$. Now the last statement follows, since a closed path leads back to the same process with the same o-number.

(2) As described above, the absence of forbidden actions can be checked when constructing $\mathsf{RTS}(P)$. In a depth-first search, which can be integrated into this construction, we can assign the prospective o-numbers by assigning 0 to P and continuing as described in (1). According to (1), P is not a response process, if due to two different transitions we try to assign different numbers to the same process. So assume that this never happens.

If some assigned o-number is negative, then by construction there is some $w \in \mathsf{RT}(P)$ with too many *out*'s—hence P is not a response process—, and vice versa. Otherwise, we know that no reachable Q can perform more *out*'s than its assigned o-number.

To finish the proof of functional correctness, which also shows that the assigned o-numbers are correct, we have to check that each reachable Q can perform enough *out*'s; i.e., from each reachable Q, there must be a path without *in*'s that reaches some of the processes with assigned o-number 0. This is a check of backwards reachability from the set of these processes, which can be done in linear time e.g. by breadth-first search or by integrating it into the backtracking of the depth-first search that assigns the o-numbers. □

3.3. Results on the response performance

We call a function f from $\mathbb{N}$ to $\mathbb{N}_0$ *asymptotically linear*, if there are constants $a, c \in \mathbb{R}$ such that $an - c \leqslant f(n) \leqslant an + c$ for all $n \in \mathbb{N}$; we call a the *asymptotic factor* of such a

function. Observe that our notion is quite strict, since f is also bounded from below and it is not only $an + o(n)$, but actually $an + O(1)$ and, more precisely, $an + \Theta(1)$. We will prove that the response performance of a finite-state response process P is asymptotically linear, and we will show how to determine its asymptotic factor, which we call the *asymptotic performance* of P.

To find out about the response performance of a response process P, it will turn out to be sufficient to consider specific paths in a reduced version of $\mathsf{RTS}(P)$.

Definition 13. For a response process P, the *reduced refusal transition system* $\mathsf{rRTS}(P)$ of P is obtained from $\mathsf{RTS}(P)$ as follows: we keep all action transitions, but we keep time steps $Q \xrightarrow{X}_r Q'$ only if either the refusal set X is $\mathbb{A}$ or $\neg Q \xrightarrow{\mathbb{A}}_r Q'$, X is $\{out\}$ and the o-number of Q is positive; then, we delete all processes that are not reachable anymore.

We call a path in $\mathsf{rRTS}(P)$ *n-critical*, if it contains at most n *in*'s and at most $n-1$ *out*'s and all time steps before the nth *in* are full.

Based on rRTS, we now give a graph–theoretical characterization for the response performance of a response process.

Theorem 14. *The response performance $rp_P(n)$ of a response process P is the supremum of the numbers of time steps taken over all n-critical paths.*

Proof. To determine $rp_P(n)$, we have to consider paths in $\mathsf{TTS}(P \parallel U_n)$ not containing ω and to count their numbers of full time steps; these are just the paths in $\mathsf{RTS}(P \parallel U_n)$ that do not contain ω and only contain time steps that are full. Clearly, such a path can have at most n *in*'s and at most n *out*'s. But after the nth *out*, an urgent ω becomes enabled and no further full time step can occur before ω. Thus, we can restrict attention to paths satisfying the condition of having at most n *in*'s and at most $n-1$ *out*'s—and therefore no ω.

We first show that for each path in $\mathsf{RTS}(P \parallel U_n)$ satisfying this condition and having only full time steps, there is an n-critical path with the same number of time steps. Each such path arises—according to our SOS-rules for parallel composition—from a path in $\mathsf{RTS}(P)$ and one in $\mathsf{RTS}(U_n)$ satisfying the same condition, and each full time step arises from two conditional time steps according to Rule Par_r in Definition 3. We now argue that the path in $\mathsf{RTS}(P)$ is essentially also in $\mathsf{rRTS}(P)$. Since action transitions of $\mathsf{RTS}(P)$ are preserved in $\mathsf{rRTS}(P)$, we only have to consider time steps. So consider a full time step $Q \parallel U \xrightarrow{1} Q' \parallel U'$ on the path in $\mathsf{RTS}(P \parallel U_n)$ arising from $Q \xrightarrow{X}_r Q'$ in $\mathsf{RTS}(P)$ and $U \xrightarrow{Y}_r U'$ in $\mathsf{RTS}(U_n)$. (In fact, we must have $U' \equiv U$.) Due to Par_r, it suffices to show that $Q \xrightarrow{X'}_r Q'$ is in $\mathsf{rRTS}(P)$ for some $X' \supseteq X$.

As argued above, X can be (and is) assumed to be $\mathbb{A}$, $\{out\}$, $\{in\}$ or $\emptyset$. Since the paths contain at most $n-1$ *out*'s, Y cannot contain both, *in* and *out*, and thus X cannot be $\emptyset$. If X is $\{in\}$, then Y must contain *out*, and the number of *in*'s and *out*'s are equal for each path reaching U in $\mathsf{RTS}(U_n)$ or Q in $\mathsf{RTS}(P)$; hence, Q cannot perform *out* (P is a response process), i.e. it can additionally refuse *out*, and by Proposition 5.2 we can replace X by $\mathbb{A}$; now $Q \xrightarrow{\mathbb{A}}_r Q'$ is also in $\mathsf{rRTS}(P)$.

If X is $\mathbb{A}$, $Q \xrightarrow{X}_r Q'$ is clearly also in $\mathsf{rRTS}(P)$. Finally, if X is $\{out\}$, then Y must contain *in*, i.e. on the paths to Q and U we have n *in*'s, but fewer *out*'s as argued above. Thus, the o-number of Q is positive, and again $Q \xrightarrow{X}_r Q'$ is also in $\mathsf{rRTS}(P)$. This shows that essentially the path in $\mathsf{RTS}(P)$ that we considered can be found in $\mathsf{rRTS}(P)$ as well and satisfies the definition of an n-critical path.

These considerations have shown that there is an n-critical path with at least $rp_P(n)$ time steps. It remains to show that each n-critical path can be combined with a path in $\mathsf{RTS}(U_n)$ without ω such that all time steps become full. Since P is a response process, all actions on such a path are *in*, *out* or τ, and at every stage of the path the number of *out*'s does not exceed the number of *in*'s. Hence, as far as actions are concerned there is (up to permutation of the components of U_n) a unique path in $\mathsf{RTS}(U_n)$ that can be combined (or synchronized) with the n-critical path under consideration. Since time steps do not change any process in $\mathsf{RTS}(U_n)$, the path is indeed unique (up to permutation). Each process on the path in $\mathsf{RTS}(U_n)$ can refuse ω; and if such a process is reached with n *in*'s (and possibly some *out*'s), then it can also refuse *in*. Thus, the time steps on the n-critical path can be combined with time steps in $\mathsf{RTS}(U_n)$ to get full time steps according to Rule Par_r in Definition 3. □

Observe that, since the difference between the numbers of *in*'s and *out*'s on a path is bounded by the largest o-number, time steps $\{out\}$ can only appear—intuitively speaking—at the very end of an n-critical path if n is large. Also, if n is large compared to the number of processes in $\mathsf{rRTS}(P)$, an n-critical path with many time steps must contain cycles; it turns out to be essential to find the worst cycles.

Definition 15. If a cycle in $\mathsf{rRTS}(P)$ for a response process P contains a positive number of time steps but no *in*'s (and hence no *out*'s by Theorem 12.1), we call it *catastrophic*. For P without catastrophic cycles, we consider cycles which can be reached from P by a path where all time steps are full and which themselves contain only time steps that are full; we define the *average performance* of such a cycle as the number of its full time steps divided by the number of *in*'s on the cycle, and we call a cycle *bad*, if it is a cycle of maximal average performance in $\mathsf{rRTS}(P)$.

The following theorem gives a graph–theoretical characterization of correct response processes and their asymptotic performance. Results on the costs for deciding correctness, computing the asymptotic performance, resp., are given in the theorem after.

Theorem 16 (*Bad-cycle theorem*). *Let P be a finite-state response process. P has a catastrophic cycle if and only if its response performance is ∞ for some n, i.e. if and only if it is not correct. If P is correct, the response performance is asymptotically linear, and the asymptotic performance of P is the average performance of a bad cycle.*

Proof. First, assume that P has a catastrophic cycle. Let m be the number of *in*'s on a path in $\mathsf{rRTS}(P)$ from P to this cycle; by definition of $\mathsf{rRTS}(P)$, either all time steps of the cycle are full (we choose $n = m + 1$ to make sure that there are less than n *out*'s on the

path) or otherwise all processes on the cycle have the same positive o-number, the path to the cycle has less than m *out*'s and we choose $n = m$. We will show that, for each k, there is an n-critical path with at least k time steps.

If we use the path and then k times the catastrophic cycle, this corresponds to a refusal trace uv^k that contains at most n *in*'s and less than n *out*'s—all of which are in the u-part—, and if v contains some non-full time step, then u contains n *in*'s. Thus, we have constructed a path with at least k time steps, which is n-critical unless u contains some non-full time step. In the latter case, we repeatedly apply Proposition 5.3 to uv^k and each of these non-full time steps in u, obtaining some $u'v^k \in \mathsf{RT}(P)$ that corresponds to an n-critical path with at least k time steps. Hence, the response performance is ∞ for n.

Second, assume that the response performance is ∞ for some n. Let r be the number of processes in $\mathsf{rRTS}(P)$ and consider an n-critical path with at least $r(n+1)$ time steps, which exists by assumption. We can subdivide this path into $n+1$ subpaths with at least r time steps each. On each subpath, there are at least $r+1$ processes where a time step starts or ends; thus, we must have a repetition of a process, i.e. a cycle containing at least one time step. Among these (not necessarily different) $n+1$ cycles, there must be one not containing any *in*, thus being catastrophic.

Now we assume that there is no catastrophic cycle; hence, $rp_P(n)$ is always finite, and a bad cycle exists. (For the latter, recall that there are only finitely many cycles; as a consequence of Proposition 5.4, there is at least one.) Let a be the average performance of such a cycle, k the number of *in*'s on it and m the number of *in*'s on a path to it that contains only full time steps. By Theorem 14, we have to show that for large n the maximal number of time steps on an n-critical path is an up to a constant. To bound this number from below, we consider $n > m + k$ and construct an n-critical path by taking the path to the cycle and running round the cycle as often as possible without getting more than n *in*'s. This path has m *in*'s in the initial part; then, it runs completely round the cycle several times; and finally, it might start a round without completing it, and this final part contains less than k *in*'s. This ensures that at least $n - m - k$ *in*-transitions are passed on completed cycles, which together have $a(n - m - k)$ time steps; thus, the response performance of P for n is at least $an - a(m + k)$, where $a(m + k)$ is a constant.

It remains to bound the number of time steps on an n-critical path from above; so consider some n-critical path. We subdivide it into the initial part and the rest path, which starts after the nth *in* if it exists, and is empty if there are less than n *in*'s.

In the rest path, there are no actions *in* and, thus, there are no cycles containing time steps—such a cycle would be catastrophic. Thus, the rest path contains at most c_1 time steps, where c_1 is a constant bound on time steps in paths containing neither *in* nor cycles.

Now we transform the initial part $P \equiv P_0 P_1 \ldots$ into cycles (which have no repetitions of processes apart from the last process) and a path from P without repetition of a process as follows: If P_j is the first process on the path that already occurred before, say as P_i, then remove the subpath $P_i \ldots P_j$—which is a cycle. Apply this to the remaining path $P_0 \ldots P_i \equiv P_j \ldots$ as often as possible.

By choice of a, the overall number of time steps on the cycles is at most an. The path that remained in the end has no cycles, starts at P and has only full time steps; there clearly is a constant c_2 bounding the number of time steps on such a path.

Together, the original path contains at most $an + c_1 + c_2$ time steps. This shows that the response performance of P is asymptotically linear (choose the required constant c as $\max(a(m+k), c_1 + c_2)$) with factor a. □

It has to be remarked that results about asymptotic linearity and its relation to the average performance of cycles is not unusual in the area of performance evaluation; such results can often be studied in the framework of $(max, +)$-algebras [4]. In a more standard formulation, one would describe a system by a directed graph, where each edge represents some task and is labelled with some cost or with its execution time; then, one would be interested in the quotient of the sum of times over the number of edges in a cycle.

The important differences to this standard situation are: the transition system $\mathsf{RTS}(P)$ generally used in timed testing contains different types of time steps; we have shown how to reduce these to just two types ($\mathsf{rRTS}(P)$) in our simple, but relevant testing scenario focussed on a restricted class of tests and then how to determine the response performance from this reduction graph–theoretically (n-critical paths); for some correctness feature, we have defined catastrophic cycles as a graph–theoretic characterization; finally, to determine the asymptotic performance of correct processes with bad cycles, we have shown that an almost standard graph with just one type of time step suffices. The non-standard feature of having edges that are 'not productive' and/or no time steps will be dealt with in the next proof.

If $\mathsf{rRTS}(P)$ is finite, then it has only finitely many cycles; hence, by Theorem 16, it can be decided whether the response performance of a finite-state response process is always finite, and if so, the asymptotic performance can be computed. The following theorem shows that both problems can be solved in reasonable time. For the second part of the following theorem, we are indebted to Torben Hagerup who pointed out the relevant paper to us and provided us with a translation of our problem to the standard problem.

Theorem 17. *Let P be a finite-state response process and n be the number of processes in $\mathsf{rRTS}(P)$. It can be decided in time $O(n^3)$ whether P has a catastrophic cycle. If no catastrophic cycle exists, the average performance of a bad cycle can be computed in time $O(n^3)$.*

Proof. To check for a catastrophic cycle, delete all *in*- and *out*-labelled arcs in $\mathsf{rRTS}(P)$, give time steps length -1 and all other arcs length 0. Then run the Floyd–Warshall algorithm for shortest paths; there exists a catastrophic cycle if and only if some entry on the diagonal of the resulting matrix is negative.

Now assume that no catastrophic cycle exists. We will transform the problem to compute the average performance of a bad cycle in time $O(n^3)$ to a similar problem in the standard formulation as indicated above, which can be solved in $O(n^3)$ as well. Clearly, we can delete all non-full time steps in $\mathsf{rRTS}(P)$ and all processes that are then not reachable anymore; call the resulting arc-labelled graph G. To get closer to an algorithm known from literature, we will compute the average throughput in G, which is just the inverse of the average performance: the *average throughput* of a cycle is the number of *in*'s divided by the number of (here necessarily full) time steps; it is ∞ if the latter is 0. Thus, we want to determine the minimal average throughput of a cycle. Since processes do not have time-stops, P can

perform arbitrarily many time steps; thus, there must be a cycle containing some time step, and the minimal average throughput is certainly finite.

We must now transform G to a graph where each arc represents one time step; such an arc will correspond to a path with one time step in G. Let G_0 be obtained from G by deleting all time steps. Let $d(u, v)$ be the length of a shortest path from u to v in G_0, where the length of an *in*-transition is 1 and all other arcs have length 0; these values can be determined in time $O(n^3)$ with the Floyd–Warshall algorithm. Now we construct a graph G' on the vertices of G as follows: whenever $d(u, v)$ is finite and there is a time step from v to v', we insert an arc uv' (which is possibly a loop) with cost $d(u, v)$; if there are several cost values for the same arc, we take the minimum. This construction including the computation of the $d(u, v)$ can still be carried out in time $O(n^3)$.

If we define the mean cost of a cycle in G' as the sum of costs divided by the number of edges, we get the following connection: a cycle in G with minimal finite average throughput t can be subdivided into paths, each ending with its only time step; each of these corresponds to an arc in G', and thus the cycle corresponds to a cycle in G' with mean cost t. Vice versa, a cycle in G' with minimal mean cost t corresponds to a closed path with average throughput t; this closed path can be seen as a union of cycles, such that each of them has an average throughput of at least t according to the previous sentence. Since these cycles altogether have average throughput t, we find among them a cycle with average throughput t.

Hence, it suffices to compute the minimal mean cost of a cycle in G'; disregarding the presence of loops, which can easily be treated separately, this can be done in $O(n^3)$, see [6]. □

Observe that Theorem 14 shows that the response performance of a response process P only depends on the language of $\mathsf{rRTS}(P)$ (defined as in Definition 4). Therefore, for all our considerations, we can replace $\mathsf{rRTS}(P)$ by any transition system with the same language (provided it is finite if $\mathsf{rRTS}(P)$ is). Theorems 16 and 17 apply just as well if we use a finite transition system language-equivalent to $\mathsf{rRTS}(P)$—even if P itself is not finite state. We specifically note the case of bisimulation quotients, which are useful for the examples in the next section.

As usual, a *bisimulation* on $\mathsf{rRTS}(P)$ is a relation $\mathcal{R}$ between processes in $\mathsf{rRTS}(P)$, such that $(Q, R) \in \mathcal{R}$ implies

- If in $\mathsf{rRTS}(P)$ $Q \xrightarrow{\alpha} Q'$, $Q \xrightarrow{X}_r Q'$ resp., then $R \xrightarrow{\alpha} R'$, $R \xrightarrow{X}_r R'$ resp., for some R' with $(Q', R') \in \mathcal{R}$,
- vice versa.

If some bisimulation is an equivalence, the respective *bisimulation quotient* is a graph with equivalence classes $[Q]$ as vertices that has an α- or X-labelled arc from $[Q]$ to $[Q']$ whenever $Q \xrightarrow{\alpha} Q'$, $Q \xrightarrow{X}_r Q'$, respectively. The definitions of catastrophic or bad cycle and average performance carry over to bisimulation quotients. (Note that all processes in some $[Q]$ have the same o-number in $\mathsf{rRTS}(P)$, which is also the o-number of $[Q]$ in the quotient.)

Proposition 18. *For a finite-state response process P and a bisimulation quotient T of $\mathsf{rRTS}(P)$, P has a catastrophic cycle if and only if T has some. The average performance of a bad cycle of P is the same as the average performance of a bad cycle in T.*

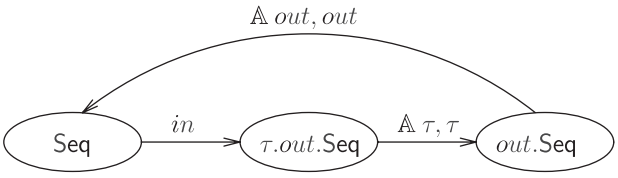

Fig. 1. The reduced refusal transition system of Seq.

4. Examples

In this section, we will apply Theorem 16 to some examples, some of which have been checked with a prototype implementation of our algorithms. We start with

$$\mathsf{Seq} \equiv \mu x.\underline{in}.\tau.out.x \quad \text{and} \quad \mathsf{Pipe} \equiv ((\mu x.\underline{in}.s.x) \|_{\{s\}} (\mu x.\underline{s}.out.x))/s$$

from the introduction. Discussing this example, we will also address the question whether our approach gives a precongruence.

Fig. 1 shows rRTS(Seq) with the simplification that e.g. the arc from $out.$Seq to Seq indicates that $out.\mathsf{Seq} \xrightarrow{out}_r \mathsf{Seq}$ and $out.\mathsf{Seq} \xrightarrow{\mathbb{A}\, out}_r \mathsf{Seq}$, where in the latter case we have left the intermediate process implicit. We will apply the same conventions in the following examples. Observe that RTS(Seq) would have an additional time step $\{out\}$ from Seq to $\underline{in}.\tau.out.$Seq (but the o-number of Seq is 0) and an additional time step $\{in\}$ from the omitted intermediate process just mentioned.

It is obvious that there is no catastrophic cycle and that the asymptotic performance of Seq is 2 by the bad-cycle theorem, and it is also clear from Theorem 14 that the response performance satisfies $rp_{Seq}(n) = 2n$.

With $L \equiv \mu x.\underline{in}.s.x$ and $R \equiv \mu x.\underline{s}.out.x$, we can write Pipe as $(L \|_{\{s\}} R)/s$ and each reachable process has this structure of a parallel composition to which hiding is applied; therefore, when showing rRTS(Pipe) in Fig. 2, we describe each process just by listing the two components. Also, we identify a recursive process with its unfolding by Proposition 18; e.g. the arc labelled $\{out\}$ should really lead to a process where the first component is $\underline{in}.s.L$ instead of L. Again we will apply the same conventions in the following examples.

There is no catastrophic cycle, and one can see that the asymptotic performance of Pipe is 1 by the bad-cycle theorem. Observe that a bad cycle cannot use the time step $\{out\}$ and therefore neither the two processes in the lower left corner. We have convinced ourselves that the response performance satisfies $rp_{Pipe}(n) = n+1$; to see that it is not better, consider e.g. the n-critical path that runs to $(L, out.R)$ via $(\underline{s}.L, R)$, then repeats the bad cycle that uses the *in*-labelled transition from there, runs through $(\underline{s}.L, R)$, and takes the time step $\{out\}$ after the $(n-1)$th *out*. On this path, each *in* is followed by a time step, and additionally there is the time step $\{out\}$ in the end.

To compare the efficiency of response processes in our approach, we can compare either their response or their asymptotic performances, where the latter gives a coarser relation. Both preorders are defined on the basis of our restricted class of users; therefore, as discussed in the introduction, we cannot expect them to be precongruences—in contrast to the testing preorders one usually considers.

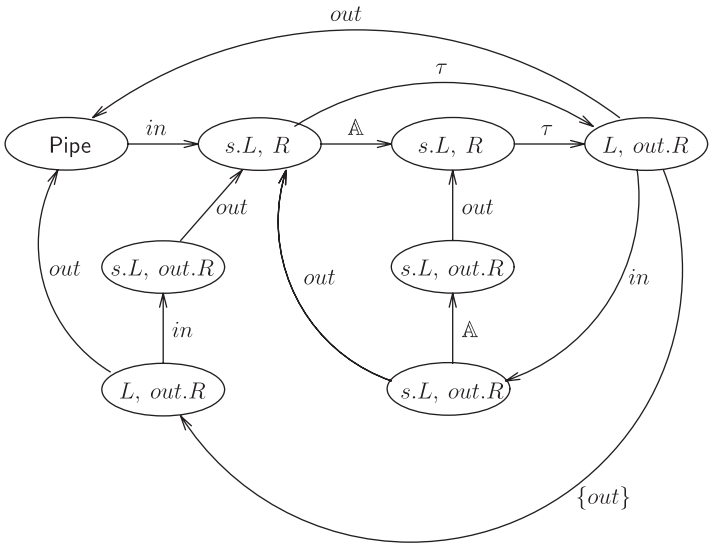

Fig. 2. The reduced refusal transition system of Pipe.

With the above example, we can show that the two preorders indeed fail to be precongruences. Consider the process $P \equiv \mu x.\underline{in}.(\underline{in}.\tau.\tau.\tau.\tau.\underline{out}.\underline{out}.x + \underline{out}.x)$; one can observe that P, $\mathsf{Seq}\|_{\{in,out\}} P$ and $\mathsf{Pipe}\|_{\{in,out\}} P$ are also response processes. It is easy to see that $rp_{Seq\|_{\{in,out\}} P}(n) = rp_{Seq}(n) = 2n$, since here the summand with the four τ's is irrelevant. In contrast, for Pipe this summand has the effect that $\mathsf{Pipe}\|_{\{in,out\}} P$ has runs where the actions *in* are alternatingly followed by either a full time step or four full time steps and two *out*'s. Hence, we have $rp_{Pipe} \leqslant rp_{Seq}$ and Pipe even has a strictly smaller asymptotic performance than Seq, while $\mathsf{Seq}\|_{\{in,out\}} P$ has with 2 a strictly smaller asymptotic performance than $\mathsf{Pipe}\|_{\{in,out\}} P$, which is (at least) 5. Thus, $rp_{Pipe\|_{\{in,out\}} P}$ cannot be less than $rp_{Seq\|_{\{in,out\}} P}$, and actually the reverse relationship holds.

This result might be unfortunate, since it disallows compositional or axiomatic reasoning. Given the practical motivation for our approach, we have to accept the situation as it is; furthermore, observe that the components of a response process will very often not be response processes themselves such that compositional reasoning is not possible anyway. In the light of the present discussion, it is also no surprise that our results are not algebraic, but have some transition systems as a basis. Nevertheless, we work in our approach with concurrent systems and concurrency is essential for performance evaluation. Thus, some mechanism to describe concurrency is needed, and we have chosen a process algebra; this seems particularly appropriate, since we follow a testing approach which essentially needs parallel composition.

One might get the impression that the response performance might actually be a linear function in all cases. To refute this, consider the process $P \equiv \mu x.\underline{in}.out.\underline{in}.\underline{out}.\underline{in}.\tau.out.x$; here $rp_P(3n-2) = 3n-2$, $rp_P(3n-1) = 3n-2$ and $rp_P(3n) = 3n$.

This might also be the right place to discuss a limitation of our approach to response performance. We have restricted consideration to the case that each user will make all

requests available from the very beginning; this can be seen as covering also cases where not all requests are available immediately, but where always as many are available as the system under test can process—until all requests have been issued. We believe that this is a very common situation. But clearly, the assumption made should be checked whenever our approach is followed.

The following example demonstrates what might go wrong if the assumption is not satisfied. Consider the process

$$B \equiv (B_1 \|_{\{s\}} B_1)/s, \quad \text{where } B_1 \equiv \mu x.\underline{in}.\tau.\tau.out.\underline{s}.x.$$

The idea of B is that two requests are *b*undled if possible; imagine that, whenever a request is received, a machine is reserved that can process another request in parallel if this is received in time. In principle, two requests can be performed as fast as one; then, work on a new bundle starts.

As above with Pipe, we note that every reachable process is a hiding applied to a parallel composition, and again we will only write the process as a pair of components. It should be clear, that no catastrophic cycle exists: B has *o*-number 0 and cannot refuse $\mathbb{A}$, so in $\mathsf{rRTS}(B)$ B must perform an *in*; as time proceeds, the corresponding τ's will be performed; eventually, *out* is performed, since otherwise it becomes urgent and no time step is possible in $\mathsf{rRTS}(B)$. If the other *in* has not been performed yet, we have reached $(\underline{s}.B_1, B_1)$—a process with *o*-number 0—and *in* must be performed now. With the same argument, we will reach $(\underline{s}.B_1, \underline{s}.B_1)$ and only now a cycle is closed with performing *s* back to *B*.

Similarly, a bad cycle is essentially

$$\begin{aligned} B &\xrightarrow{in\,in}_r (\tau.\tau.out.\underline{s}.B_1, \tau.\tau.out.\underline{s}.B_1) \xrightarrow{\mathbb{A}\tau\tau}_r (\tau.out.\underline{s}.B_1, \tau.out.\underline{s}.B_1) \\ &\xrightarrow{\mathbb{A}\tau\tau}_r (out.\underline{s}.B_1, out.\underline{s}.B_1) \xrightarrow{\mathbb{A}\,out\,out}_r (\underline{s}.B_1, \underline{s}.B_1) \xrightarrow{\tau}_r B, \end{aligned}$$

the asymptotic performance is therefore 1.5 and B is better than Seq.

Now consider a user like $\underline{in}.\underline{out}.\underline{in}.\underline{out}.\underline{in}.\underline{out}.\underline{\omega}$ that will issue the next request only if a response for the previous one has been received—the sort of user we are *not* dealing with in our approach. This user can always refuse *in* when waiting for an *out*, hence a time step $\{out\}$ by the system is sufficient for a bad performance; thus, the following behaviour becomes relevant (which is not a 3-critical path):

$$\begin{aligned} B &\xrightarrow{in}_r (\tau.\tau.out.\underline{s}.B_1, B_1) \xrightarrow{\{out\}\tau}_r (\tau.out.\underline{s}.B_1, B_1) \xrightarrow{\{out\}\tau}_r (out.\underline{s}.B_1, B_1) \\ &\xrightarrow{\{out\}\,out}_r (\underline{s}.B_1, B_1) \xrightarrow{in}_r (\underline{s}.B_1, \tau.\tau.out.\underline{s}.B_1) \xrightarrow{\{out\}\tau}_r (\underline{s}.B_1, \tau.out.\underline{s}.B_1), \text{ etc.} \end{aligned}$$

Hence, each response will take time 3, and thus B is worse than Seq for this sort of user that does not satisfy our assumption.

In all the above cases, the worst case behaviour can be obtained by synchronous behaviour, as described after Definition 2. This might give the impression that worst case behaviour of a process always means that the single actions show their full delay and therefore behave as if the process had a global clock completely governing all the actions. We now come to an example demonstrating that this is not true.

The example system 2Line is built from a scheduler S_1 that accepts actions *in* and distributes them alternatingly to two processes (production lines) P_1 and Q_1 with actions

in_1 and in_2. These processes compete for a common resource R_1 which they acquire with a and b, resp.; after using the resource they perform *out*. We define

$$\begin{aligned} S_1 &\equiv \mu x.\underline{in}.in_1.\underline{in}.in_2.x, \\ P_1 &\equiv \mu x.in_1.a.\underline{out}.x, \\ R_1 &\equiv \mu x.a.x + b.x. \end{aligned}$$

The reachable processes are: $S_2 \equiv in_1.S_4$, $S_3 \equiv \underline{in_1}.S_4$, $S_4 \equiv \underline{in}.S_5$, $S_5 \equiv in_2.S_1$ and $S_6 \equiv \underline{in_2}.S_1$; $P_2 \equiv \underline{in_1}.P_3$, $P_3 \equiv a.P_5$, $P_4 \equiv \underline{a}.P_5$ and $P_5 \equiv \underline{out}.P_1$; $R_2 \equiv \underline{a}.R_1 + \underline{b}.R_1$.

The processes Q_i are defined as the P_i with in_1 and a replaced by in_2 and b. The system is defined by

$$\mathsf{2Line} \equiv (S_1 \parallel_{\{in_1,in_2\}} ((P_1 \parallel_{\emptyset} Q_1) \parallel_{\{a,b\}} R_1)/\{a,b\})/\{in_1,in_2\}.$$

Each reachable process of 2Line can be written as a sequence of four digits, each giving the index of the component ordered as SPQR;[2] so 2Line itself corresponds to 1111.

Consider an n-critical path of 2Line, where actions are performed in synchronous mode, i.e. only if they are urgent. For large enough n, such a path starts (up to permutation of actions that occur at the same time) as follows—where for clarity we write in_1 and later a, b and in_2 although these actions are hidden such that technically these are really τ's:

$$1111 \xrightarrow{in}_r 2111 \xrightarrow{\mathbb{A}}_r 3222 \xrightarrow{in_1\,in}_r 5322 \xrightarrow{\mathbb{A}}_r 6422.$$

From here it begins to circle like this

$$6422 \xrightarrow{a\,in_2}_r 1531 \xrightarrow{in\,out}_r 2131 \xrightarrow{\mathbb{A}}_r 3242 \xrightarrow{b\,in_1}_r 4351 \xrightarrow{in\,out}_r 5311 \xrightarrow{\mathbb{A}}_r 6422.$$

So it might look like the asymptotic performance is $\frac{2}{2} = 1$. But it actually is larger, namely $\geqslant 1.5$ as we demonstrate with this cycle

$$6422 \xrightarrow{in_2\,b}_r 1451 \xrightarrow{in\,out}_r 2411 \xrightarrow{\mathbb{A}}_r 3422 \xrightarrow{a\,out}_r 3121 \xrightarrow{\mathbb{A}}_r 3222 \xrightarrow{in_1\,in}_r 5322 \xrightarrow{\mathbb{A}}_r 6422.$$

Here, without any delay, the second production line grabs the resource with b, and this way the nice coordination that we would have in synchronous mode is disturbed.

The performance can be improved by using R_1' in place of R_1: $R_1' \equiv \mu x.a.b.x$ has reachable processes $R_2' \equiv \underline{a}.R_3'$, $R_3' \equiv b.R_1'$ and $R_4' \equiv \underline{b}.R_1'$. Then, we define 2Line$'$ just as 2Line, but with R_1' in place of R_1. Reachable processes can still be described with four digits.

To study the performance of rRTS(2Line$'$), we will not build rRTS(2Line$'$), but will instead immediately build a bisimulation quotient. Observe that the reachable processes have a symmetry: if we have a reachable process and add 3 to the index of S (modulo 6), exchange the indices of P and Q and add 2 to the index of R' (modulo 4), then we get a reachable process that is bisimilar to the first one.

The following table lists the equivalence classes of a bisimulation. On the left, we give a consecutive numbering of the classes; in the middle, we list some processes, and on the

[2] Yes, one of the authors worked in Rome for a while.

right the symmetric ones. Sometimes, an index in a fourtuple is replaced by a list of possible choices, so 5,6 3,4 1 1 stands for 5311, 6311, 5411 and 6411.

Eq. classes	S_i	P_j	Q_k	R_l	$S_{(i+3) \bmod 6}$	P_k	Q_j	$R_{(l+2) \bmod 4}$
1	1	1,2	1	1	4	1	1,2	3
2	2	1,2	1	1	5	1	1,2	3
	3	1	1	1	6	1	1	3
3	3	2	2	2	6	2	2	4
4	4	3,4	1	1	1	1	3,4	3
	4	3	2	2	1	2	3	4
5	5,6	3,4	1	1	2,3	1	3,4	3
	5	3	2	2	2	2	3	4
6	4	5	1,2	3	1	1,2	5	1
7	5	1	1,2	3	2	1,2	1	1
	6	1	1	3	3	1	1	1
8	6	1	2	3	3	2	1	1
	6	2	2	4	3	2	2	2
9	1	3,4	3	1	4	3	3,4	3
10	5	5	1,2	3	2	1,2	5	1
	6	5	1	3	3	1	5	1
11	6	4	2	2	3	2	4	4
12	1	4	3,4	2	4	3,4	4	4
13	1	5	3,4	3	4	3,4	5	1
14	6	5	2	3	3	2	5	1
15	2	3,4	3	1	5	3	3,4	3
16	2,3	4	3,4	2	5,6	3,4	4	4
17	2,3	5	3,4	3	5,6	3,4	5	1
18	1	5	5	1	4	5	5	3
19	2,3	5	5	1	5,6	5	5	3
20	1	5	1	1	4	1	5	3
21	2,3	5	1	1	5,6	1	5	3
22	4	4	2	2	1	2	4	4
23	5	4	2	2	2	2	4	4

Fig. 3 shows the bisimulation quotient; since there are no catastrophic cycles, we have omitted arcs labelled $\{out\}$ and vertices only reachable via such arcs, namely vertices 22 and 23. To ease understanding, we again have not labelled arcs with τ but instead with actions a, in_1, etc. that are actually hidden. These actions can be performed by the processes listed in the middle, while the symmetric processes on the right can perform the symmetric actions instead, i.e. b, in_2, etc.

To find the bad cycles in this quotient, we proceed as follows. Assume there is a group of vertices without a cycle, and that there is only one vertex v outside the group that has arcs leading to a vertex in the group. (As an example, consider {9, 11, 12, 14, 15, 16}.) Then we can contract this group, i.e. replace it by arcs from v to the vertices w that are the target of

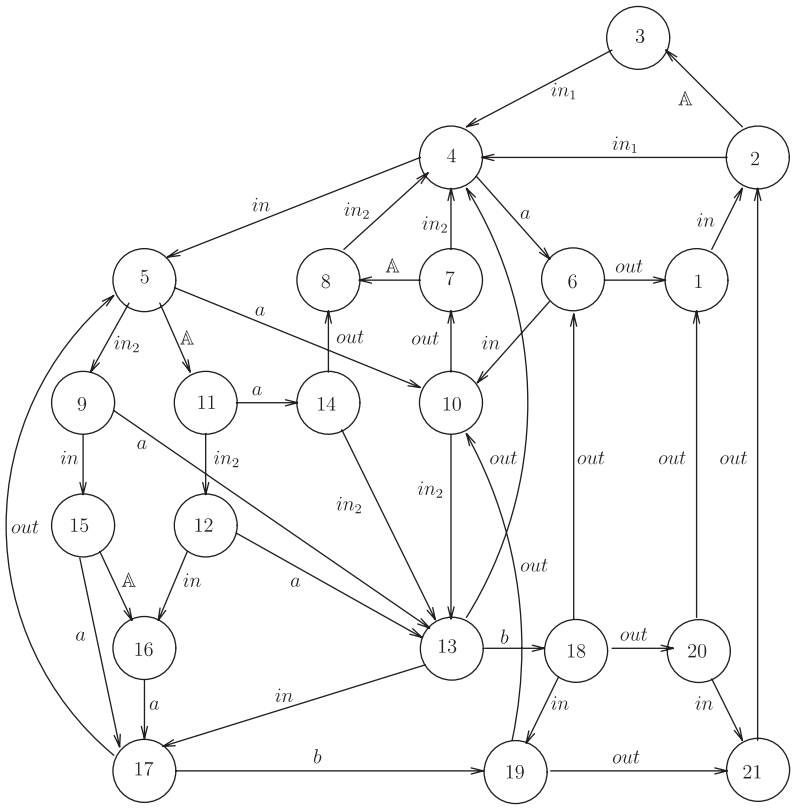

Fig. 3. The bisimulation quotient of rRTS(2Line′).

arcs from vertices in the group—provided there is a path from v through the group to w; we label such an arc from v to some w by the inscriptions of all such paths. This time, we omit all τ's and write *i* instead of *in* and *o* instead of *out*; further, we regard the ordering in such an inscription as immaterial, i.e. $\mathbb{A}o$ and $o\mathbb{A}$ are identified for example.

Since each cycle using a vertex of the group must pass through v, we can find a cycle with the same average performance in the contracted graph using one of the new arcs. Vice versa, each cycle using a new arc can be traced back to a cycle in the original graph with the same average performance. Cycles not using a vertex of the group, a new arc, resp., are of course preserved. Hence, we can check the contracted graph for bad cycles instead.

Dually, we can also perform a contraction if all arcs leaving a group go to the same vertex; consider e.g. $\{7, 8\}$. As a further simplification, we will only consider the 'worst' inscriptions; e.g. when contracting $\{7, 8\}$, the new arc from 10 to 4 will only be labelled $o\mathbb{A}$ instead of o, $o\mathbb{A}$.

Fig. 4 shows the bisimulation quotient after contracting $\{1, 2, 3, 20, 21\}$, $\{7, 8\}$ and $\{9, 11, 12, 14, 15, 16\}$.

Now observe that the arc from 5 to 10 is redundant, since there are already arcs from 5 to 4 with label $\mathbb{A}o$ and to 13 with the worse label $\mathbb{A}$. Hence, we can remove this arc

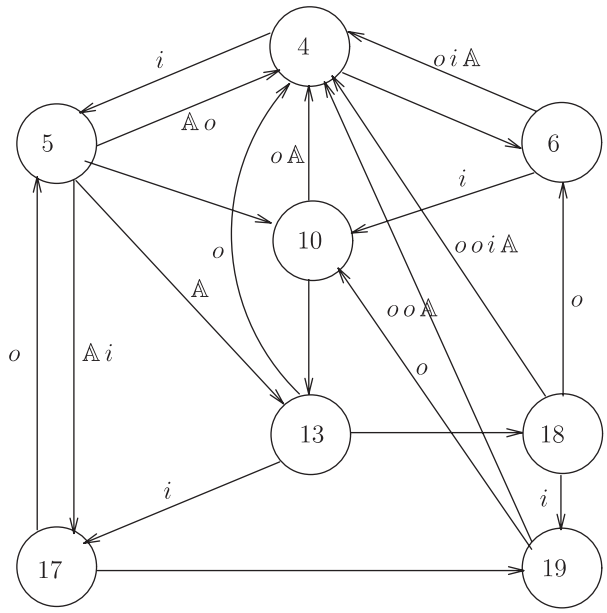

Fig. 4. Bisimulation quotient of rRTS(2Line′): first contraction.

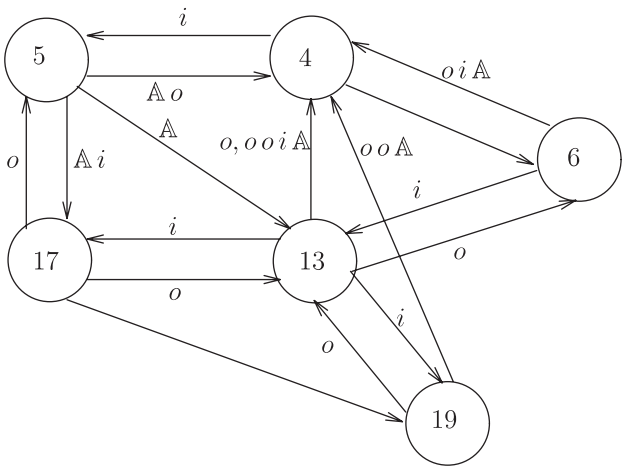

Fig. 5. Bisimulation quotient of rRTS(2Line′): second contraction.

without changing the refusal traces. (This is of course not a contraction.) Afterwards, we can contract 10 and also 18, which results in Fig. 5.

Now the arc from 13 to 19 is redundant, since going directly back to 13 does certainly not give a bad cycle and there already is an $ooi\mathbb{A}$-labelled arc from 13 to 4. Omitting the

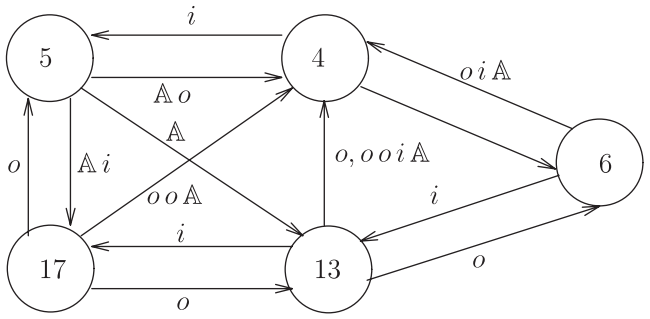

Fig. 6. Bisimulation quotient of rRTS(2Line′): third contraction.

arc from 13 to 19 and then contracting 19 results in Fig. 6. With some care, one can see that a bad cycle in Fig. 6 has average performance 1, so we conclude that this is the asymptotic performance of 2Line′.

Therefore, 2Line′ is indeed asymptotically better than 2Line and in fact as good as 2Line run in synchronous mode.

5. Conclusion

This paper follows a line of research about the efficiency of asynchronous systems, modelled as timed systems where activities have upper but no lower time bounds. In this line, the classical testing approach of De Nicola and Hennessy [3] has been refined to timed testing—first in a Petri net setting [12,5,1] and later in process algebra [2]—and the resulting testing preorder is a suitable faster-than relation. Recently, a corresponding bisimulation-based faster-than relation was studied in [7]. Upper time bounds have also been studied in the area of distributed algorithms; see e.g. [9], where also performance under restricted user behaviour is considered as mentioned in Section 3.1. A bisimulation-based faster-than relation for asynchronous systems using lower time bounds has been suggested in [11]; this approach has been improved very recently in [8]. Since the present paper develops our previous approach further, we refer the reader to Corradini, Vogler and Jenner [2] for a comparison of this approach with the literature, in particular on other timed process algebras.

Continuing [2], we have shown in this paper that the qualitative faster-than relation can also be given a quantitative formulation, concerned with how much faster one system is than another. We have argued that it is in some cases too demanding to require that a faster system is indeed faster for all possible users, since reasonable assumptions about user behaviour can restrict the class of relevant users. This paper is a realistic case study showing what can be achieved if assumptions about users can be made. We have introduced response processes and their response performance, which measures system performance under heavy load. It has been shown how the latter can be determined from what we call *n*-critical paths of a transition system rRTS(P), which we have defined specifically for the given situation. We have pointed out that even processes that are always *able* to perform the required responses

might fail to do so in a bounded time and that these processes P are characterized by what we call catastrophic cycles. For a process that never fails in this sense, we have shown that its response performance is asymptotically a linear function whose constant factor (the asymptotic performance of the process) is the average performance of what we call a bad cycle. We have also shown how to decide relevant features and compute the asymptotic performance in polynomial time.

It has to be remarked that results about asymptotic linearity and its relation to the average performance of cycles is not unusual in the area of performance evaluation; such results can often be studied in the framework of $(max, +)$-algebras [4]. In a more standard formulation, one would describe a system by a directed graph, where each edge represents some task and is labelled with some cost or with its execution time; then, one would be interested in the quotient of the sum of times over the number of edges in a cycle.

In our setting, a minor variation is that only some edges represent tasks and these take no time, while other edges represent time steps and yet others stand for no task and no time. The important differences are: the transition system $\mathsf{RTS}(P)$, which is relevant for timed testing in general, contains different types of time steps; we have shown how to reduce these to fewer types in our simple, but relevant testing scenario focussed on a restricted class of tests, and how to determine the performance from this 'performance graph' $\mathsf{rRTS}(P)$ graph–theoretically with n-critical paths. Finally, we have shown that for the asymptotic performance of correct processes one can work with a graph with just one type of time step to find a bad cycle.

We close with an example that demonstrates that our results depend on the particular users we studied. For $P \equiv \mu x.\underline{in}.\underline{out}.\underline{in}.\underline{out}.\underline{in}.\tau.\tau.\underline{out}.x$ and $Q \equiv \mu x.\underline{in}.\tau.\underline{out}.x$, we have $rp_P(3n+i) = 2n$ and $rp_Q(3n+i) = 3n+i$ for $n \in \mathbb{N}_0$ and $0 \leqslant i \leqslant 2$; intuitively, it should be clear that P takes less time per request and, indeed, its response performance as well as its asymptotic performance are lower. If we define U'_n by replacing each component $\underline{in}.\underline{out}.\underline{\omega}$ of U_n by $\underline{in}.\tau.\underline{out}.\underline{\omega}$ and define rp'_P based on these users, we get $rp_P(3n+i) = 4n+i$ and $rp_Q(3n+i) = 3n+i$ for $n \in \mathbb{N}_0$ and $0 \leqslant i \leqslant 2$. Thus, the comparison gives the opposite result; one cannot say that this opposite result is wrong, since also users like the U'_n exist—but certainly it is quite counter-intuitive.

This example should make clear that it is a challenging task to find other realistic assumptions on users than ours and develop comparable, strong results for the resulting testing preorders. After several such case studies, one might consider general strategies to treat such assumptions.

References

[1] E. Bihler, W. Vogler, Efficiency of token-passing MUTEX-solutions—some experiments, in: J. Desel, et al. (Eds.), Applications and Theory of Petri Nets, Lecture Notes in Computer Science, Vol. 1420, Springer, Berlin, 1998, pp. 185–204.

[2] F. Corradini, W. Vogler, L. Jenner, Comparing the worst-case efficiency of asynchronous systems with PAFAS, Acta Inform. 38 (2002) 735–792.

[3] R. De Nicola, M.C.B. Hennessy, Testing equivalence for processes, Theoret. Comput. Sci. 34 (1984) 83–133.

[4] S. Gaubert, Max Plus, Methods and applications of (max,+) linear algebra, in: R. Reischuk, et al. (Eds.), STACS 97, Lecture Notes in Computer Science, Vol. 1200, Springer, Berlin, 1997, pp. 261–282.

[5] L. Jenner, W. Vogler, Fast asynchronous systems in dense time, Theoret. Comput. Sci. 254 (2001) 379–422.
[6] R. Karp, A characterization of the minimum mean-cycle in a digraph, Discrete Math. 23 (1978) 309–311.
[7] G. Lüttgen, W. Vogler, A faster-than relation for asynchronous processes, in: K. Larsen, M. Nielsen (Eds.), CONCUR 01, Lecture Notes in Computer Science, Vol. 2154, Springer, Berlin, 2001, pp. 262–276, 'Bisimulation on Speed: Worst-Case Efficiency', Inform. Comput. 191 (2004) 105–144.
[8] G. Lüttgen, W. Vogler, Bisimulation on Speed: Lower Time Bounds, FOSSACS 04, Barcelona, März 2004, Springer, Berlin, Heidelberg, 2004, Lect. Notes Comput. Sci. 2987, 333–347.
[9] N. Lynch, Distributed Algorithms, Morgan Kaufmann Publishers, San Francisco, 1996.
[10] R. Milner, Communication and Concurrency, Prentice-Hall, Englewood Cliffs, NJ, 1989.
[11] F. Moller, C. Tofts, Relating processes with respect to speed, in: J. Baeten, J. Groote (Eds.), CONCUR '91, Lecture Notes in Computer Science, Vol. 527, Springer, Berlin, 1991, pp. 424–438.
[12] W. Vogler, Faster asynchronous systems, Inform. Comput. 184 (2003) 311–342.